MostoDE: A Tool to Exchange Data amongst Semantic-web Ontologies

Carlos R. Rivero[*], Inma Hernández, David Ruiz, Rafael Corchuelo

University of Sevilla, ETSI Informática, Avda. Reina Mercedes s/n, Sevilla E-41012, Spain

Abstract

A semantic-web ontology, simply known as ontology, comprises a data model and data that should comply with it. Due to their distributed nature, there exist a large amount of heterogeneous ontologies, and a strong need for exchanging data amongst them, i.e., populating a target ontology using data that come from one or more source ontologies. Data exchange may be implemented using correspondences that are later transformed into executable mappings; however, exchanging data amongst ontologies is not a trivial task, so tools that help software engineers to exchange data amongst ontologies are a must. In the literature, there are a number of tools to automatically generate executable mappings; unfortunately, they have some drawbacks, namely: 1) they were designed to work with nested-relational data models, which prevents them to be applied to ontologies; 2) they require their users to handcraft and maintain their executable mappings, which is not appealing; or 3) they do not attempt to identify groups of correspondences, which may easily lead to incoherent target data. In this article, we present MostoDE, a tool that assists software engineers in generating SPARQL executable mappings and exchanging data amongst ontologies. The salient features of our tool are as follows: it allows to automate the generation of executable mappings using correspondences and constraints; it integrates several systems that implement semantic-web technologies to exchange data; and it provides visual aids for helping software engineers to exchange data amongst ontologies.

Keywords: Data Exchange, Semantic Web, Executable Mappings, SPARQL

1. Introduction

The goal of the Semantic Web is to endow the current Web with metadata, i.e., to evolve it into a Web of Data that can be easily consumed by machines (Polleres and Huynh, 2009). Semantic-web ontologies are the artefacts of this Web of Data, each of which comprises a data model and data that should comply with it; they build on the semantic-web technologies, i.e., RDF, RDF Schema, and OWL for modelling structure and data, and SPARQL for querying them (Antoniou and van Harmelen, 2008; Zhang et al., 2010). For the sake of brevity, in this article, we refer to semantic-web ontologies as ontologies.

The distributed nature of the Web of Data implies that there exist a large amount of heterogeneous ontologies that have been devised by different organisations with different purposes. Furthermore, the same data in the Web of Data may be endowed with different data models that have been devised by different organisations, who use those data with different purposes; this makes it common to have interoperability problems (Noy, 2004). Furthermore, new ontologies try to reuse existing ontologies as much as possible since it is considered a good practice. Unfortunately, it is usual that existing ontologies cannot be completely reused, but require to be adapted, which entails that ontologies use a mixture of new and reused data models (Heath and Bizer, 2011). Due to these facts, there exist a strong need for exchanging data amongst ontologies, i.e., populating a target ontology using data that come from one or more source ontologies (Fagin et al., 2005).

Exchanging data amongst ontologies may be performed by means of ad-hoc proposals, which rely on handcrafting pieces of code to solve particular data exchange problems (Omelayenko, 2002), reasoner-based proposals, which rely on logic rules that express how source data can be reclassified into target data (Serafini and Tamilin, 2007), and query-based proposals, which rely on queries that retrieve data from a source ontology, transform them, and output the results using the target ontology (Rivero et al., 2011a). Both the rules required by reasoner-based proposals and the queries required by query-based proposals are usually referred to as executable mappings in the literature. They are represented in a high-level, structured language (Rivero et al., 2011a). Note that, in addition to data exchange, executable mappings are the cornerstone components of several other integration tasks, such as data integration (Lenzerini, 2002; Makris et al., 2012, 2010), model matching (Bellahsene et al., 2011), model evolution (Noy and Klein, 2004), or query processing in distributed ontologies (Lee et al., 2010).

Ad-hoc proposals are difficult to create, tune, maintain, and reuse since they require an expert to handcraft a piece of software to solve each data exchange problem independently from the others (Popa et al., 2002). Reasoner-based proposals are generally not scalable since they do not achieve good perfor-

[*]Corresponding author. Tel.: +34954552770.
Email addresses: carlosrivero@us.es (Carlos R. Rivero), inmahernandez@us.es (Inma Hernández), druiz@us.es (David Ruiz), corchu@us.es (Rafael Corchuelo)

mance when reasoning on relatively complex or large ontologies (Haarslev and Möller, 2008). These facts make the previous proposals little appealing to deal with real-world data exchange problems, which has motivated many authors to work on query-based proposals.

The work regarding query-based proposals has focused on generating executable mappings automatically. This is a key requirement since, otherwise, the costs involved in devising the mappings, checking if they work as expected, and optimising and maintaining them would not make sense (Petropoulos et al., 2007). The vast majority of proposals build on correspondences, which are hints that specify which entities in the source and target ontologies correspond to each other, i.e., are somewhat related (Euzenat and Shvaiko, 2007). The literature also provides several proposals to generate correspondences automatically (Bellahsene et al., 2011); unfortunately, they are not usually able to find all of the correspondences involved in some data exchange problems, and thus require user intervention (Raffio et al., 2008b). Additionally, correspondences are inherently ambiguous, i.e., different experts may interpret them differently and thus the same data exchange problem might result in different target data (Alexe et al., 2008b; Bernstein and Melnik, 2007).

As a conclusion, tools that help software engineers devise the queries to exchange data amongst ontologies are a must. In the literature, there are a number of tools to automatically generate executable mappings (Bizer and Schultz, 2010; Bonifati et al., 2005; Dou et al., 2005; Haas et al., 2005; Maedche et al., 2002; Mocan and Cimpian, 2007; Parreiras et al., 2008; Ressler et al., 2007). Unfortunately, these tools have a number of drawbacks: 1) Haas et al. (2005) and Bonifati et al. (2005) rely on nested-relational data models, and cannot thus be applied to ontologies. 2) Bizer and Schultz (2010), Dou et al. (2005), Parreiras et al. (2008) and Ressler et al. (2007) require their users to handcraft and maintain their executable mappings, which is not appealing since users have to create, optimise, and maintain them. 3) Mocan and Cimpian (2007) did not attempt to identify groups of correspondences, which may easily lead to target data that does not satisfy the constraints in the target ontology (Alexe et al., 2011a; Bernstein and Melnik, 2007; Popa et al., 2002).

In this article, we present MostoDE[1], a tool that assists software engineers in generating SPARQL 1.1 executable mappings and exchanging data amongst a subset of ontologies that can be represented in quite a complete subset of the OWL 2 Lite profile. These mappings are automatically generated based on correspondences and constraints amongst source and target ontologies, and they are executed by means of a query engine on the source ontologies to produce data that comply with the target ontology. Our tool provides a GUI that allows to define correspondences amongst the source and target ontologies. These correspondences are n:1, which means that it is possible to specify one or more entities of the source ontologies that correspond to one entity of the target ontology; we also allow to specify transformation functions in the correspondences. Furthermore, MostoDE also allows to define user-defined constraints in the source or target ontologies to adapt them to the requirements of a specific data exchange problem (Bouquet et al., 2004). The executable mappings output by MostoDE are represented in a subset of the SPARQL 1.1 query language that includes blank nodes and BIND clauses with user-defined functions to transform source data. Currently, there exists a variety of systems that implement semantic-web technologies and are, thus, suitable to perform data exchange, e.g., Sesame, OWLIM, Jena, TDB, Oracle, ARQ, or Pellet to mention a few; as of the time of writing this article, W3C lists a total of 282 systems (W3C, 2012). MostoDE integrates several of these systems by means of an open framework that is intended to provide extension points that software engineers can use to incorporate new systems.

The salient features of our tool are as follows: it allows to automate the generation of executable mappings using correspondences and constraints; it integrates several systems that implement semantic-web technologies to exchange data; additionally, it provides visual aids for helping software engineers exchange data amongst ontologies that are represented in quite a complete subset of the OWL 2 Lite profile (see (Rivero et al., 2012b) for further details). We presented a preliminary version of our tool in (Rivero et al., 2011b) and a formalisation of the algorithms behind MostoDE in (Rivero et al., 2012b); in this version, we have improved our previous results since now we can deal with n:1 correspondences and functions to transform data as they are exchanged. This is an important improvement insofar we can now deal with common problems in practice, e.g., concatenating a name and a surname to form a full name, or looking up a code in a database to dereference it.

The rest of the article is organised as follows: Section 2 presents the related work; Section 3 reports on some preliminaries regarding semantic-web technologies and data exchange. Section 4 describes the underlying data model and algorithms of MostoDE. Sections 5 and 6 respectively presents how we implemented our tool, and the evaluation results that we have obtained after applying it to four real-world data exchange problems. Finally, Section 7 recaps on our main conclusions.

2. Related work

In this section, we present other existing tools that are related to MostoDE; on the one hand, we present some tools that focus on nested-relational data models (see Section 2.1), and, on the other hand, we describe some tools that rely on semantic-web ontologies (see Section 2.2). Finally, we analyse and discuss the drawbacks of these tools (see Section 2.3).

Table 1 summarises this section. In this table, the ✓ symbol denotes that the tool supports a feature, symbol ✕ implies that the tool does not support a feature, symbol N/A states that this feature is not applicable to the tool, and the ∼ symbol indicates that the feature is partially supported. The features we have analysed are the following:

F_1: This feature determines if a tool allows to generate executable mappings automatically. Otherwise, the mappings

[1]`http://tdg-seville.info/carlosrivero/MostoDE`

Tools	F_1	F_2	F_3	F_4	F_5	F_6	F_7
Nested-relational tools							
(Haas et al., 2005)	✓	✓	✓	×	×	N/A	N/A
(Alexe et al., 2006)	✓	✓	✓	×	×	N/A	N/A
(Bonifati et al., 2005)	✓	✓	✓	×	×	N/A	N/A
(Alexe et al., 2008a)	✓	✓	✓	×	×	N/A	N/A
(Raffio et al., 2008a)	✓	✓	✓	✓	✓	N/A	N/A
(Mecca et al., 2009b)	✓	✓	✓	✓	✓	N/A	N/A
(Pichler and Savenkov, 2009)	×	×	✓	×	×	N/A	N/A
(Alexe et al., 2011b)	✓	×	✓	×	×	N/A	N/A
(Marnette et al., 2011)	✓	✓	✓	✓	✓	N/A	N/A
Semantic-web tools							
(Mocan and Cimpian, 2007)	✓	×	×	✓	✓	✓	~
(Maedche et al., 2002)	✓	×	✓	✓	✓	✓	×
(Parreiras et al., 2008)	×	✓	✓	×	✓	✓	~
(Bizer and Schultz, 2010)	×	✓	✓	×	✓	✓	×
(Dou et al., 2005)	×	✓	×	×	✓	✓	~
(Ressler et al., 2007)	×	✓	✓	×	✓	✓	~
MostoDE	✓	✓	✓	✓	✓	✓	~

F_1 = Automatically generate executable mappings; F_2 = Interpret correspondences in groups; F_3 = Use a query engine to exchange data; F_4 = Use n:1 correspondences; F_5 = Deal with transformation functions; F_6 = Deal with arbitrary RDFS ontologies; F_7 = Deal with arbitrary OWL ontologies.

Table 1: Comparison of tools to exchange data.

must be handcrafted, which is undesirable because this is an effort-consuming and error-prone task.

F_2**:** This feature determines if a tools interprets correspondences in groups of inter-related correspondences. Otherwise, it interprets them in isolation, which is undesirable because it can easily lead to incoherent target data.

F_3**:** This feature determines whether or not a tool performs data exchange by means of a query engine. The few proposals that are not query-based are reasoner-based, which, as we mentioned earlier, do not seem scalable enough.

F_4**:** This feature determines if a tool deals with n:1 correspondences since it may be necessary to relate more than one source entity in the same correspondence.

F_5**:** This feature determines if a tool deals with transformation functions since they are appealing when exchanging data in the context of the Web of Data.

F_6**:** This feature determines if a tool can deal with arbitrary RDFS ontologies that include constraints such as domain, range, or subclass. Otherwise, it can only deal with a subset of them, e.g., some taxonomies, which is not desirable since the current Web of Data is composed of arbitrarily complex ontologies.

F_7**:** This feature determines if a tool can deal with arbitrary OWL ontologies that include constraints, such as minimal cardinality, union of classes, or intersection of instances.

2.1. Nested-relational tools

The earliest attempts to devise a tool to generate executable mappings focused on the exchange of data amongst nested-relational data models. Haas et al. (2005) devised Clio, which is the state-of-the-art tool and lies at the heart of IBM's InfoSphere Data Architect (IBM, 2012). Clio takes a source and a target nested-relational data models, a number of constraints of each data model, and a number of 1:1 correspondences between them as input, and it is able to automatically generate executable mappings in different query languages, such as XQuery, XSLT, or SQL. Furthermore, Clio allows to exchange data from the source to the target by running the generated executable mappings on a query engine.

Alexe et al. (2006) built SPIDER on top of Clio; it is a tool that helps understand and maintain the automatically generated executable mappings by extracting data examples from the source and the target. Bonifati et al. (2005) devised HePToX, which focuses on Peer-to-Peer systems in which each peer stores its own data and can integrate data from other peers. HePToX is similar to Clio but, instead of working with nested-relational data models, it focuses on XML data models, which are a superset of nested-relational data models. In addition, HePToX allows to perform data exchange amongst the peers contained in a particular Peer-to-Peer system.

Recent proposals have focused on improving the previous tools. Alexe et al. (2008a) devised Muse to automatically generate executable mappings building on n:1 correspondences; Muse is able to automatically infer grouping functions for correspondences by analysing the answers to a sequence of questions proposed by the tool. Raffio et al. (2008a) developed Clip, which also allows to generate executable mappings based on n:1 correspondences, and it uses a mapping visual language that was specifically designed for nested-relational data models. It allows to represent complex correspondences, such as grouping functions, aggregation functions, or dependent correspondences. Mecca et al. (2009b) devised +Spicy, a tool that allows to compute core executable mappings, a special type of executable mappings that generates core target data when performing data exchange. The most appealing feature of core target data is that they are non-redundant data (Mecca et al., 2009a). Marnette et al. (2011) devised ++Spicy, which improves +Spicy by allowing more expressive target constraints.

Pichler and Savenkov (2009) developed DEMo, which also computes core executable mappings and allows to evaluate redundancy in the generated target data model of a particular data exchange problem. Alexe et al. (2011b) presented Eirene, a tool to automatically generate executable mappings using data examples instead of constraints and correspondences. Note that this tool assumes that data examples pre-exist but, if they do not exist, the user is responsible for providing them.

The industry has paid attention to the problem of exchanging data in nested-relational settings. The following range amongst the most popular tools: Carey (2006), IBM (2012), Microsoft (2012), Altova (2012), and Stylus (2012).

2.2. *Semantic-web tools*

Euzenat et al. (2008) and Polleres et al. (2007) presented preliminary ideas on the use of SPARQL 1.0 executable mappings to exchange data. They focused on the lacks of standard SPARQL to work as a language to describe executable mappings, and they proposed a number of extensions, such as regular expressions to describe paths, external functions, or aggregations. Furthermore, Bikakis et al. (2009) presented SPARQL2XQuery, which aims to answer SPARQL queries over XML databases.

Mocan and Cimpian (2007) developed the Web Services Execution Environment in which they studied the problem of data exchange in the context of semantic-web services, i.e., web services that are enriched with semantic annotations to improve their discovery and composition (Forte et al., 2008). They presented a formal framework to describe correspondences in terms of first-order logic formulae that can be mapped onto Web Service Modeling Language rules very easily. Note that the Web Service Modeling Language is an ontology language that takes into account the features identified by the Web Service Modeling Ontology (Fensel et al., 2007). Their proposal is similar in spirit to the one by Maedche et al. (2002), whose focus was on modelling correspondences in a general-purpose setting. The main difference with the previous proposals is that Mocan and Cimpian (2007) went a step beyond formalising correspondences and devised a tool that executes them using a WSML reasoner.

Parreiras et al. (2008) presented a tool within the framework of Model-Driven Engineering. They extended the ATL metamodel to support OWL ontologies, which allows to express constraints on them using the OCL language. They devised a mapping language called MBOLT by means of which users can express executable mappings that are later transformed into the SPARQL query language by means of a library of ATL transformations. This is similar in spirit to the proposals by Bizer and Schultz (2010), Dou et al. (2005), and Ressler et al. (2007), the difference is the language used to represent the executable mappings: Bizer and Schultz (2010) and Ressler et al. (2007) use SPARQL 1.0 executable mappings; whereas Dou et al. (2005) use Web-PDDL executable mappings that are run by means of a first-order logic reasoner.

Gennari et al. (2003) devised Protégé, one of the most successful tools to build and maintain ontologies. Protégé includes the PROMPT Suite (Noy and Musen, 2003) that allows to automatically find correspondences amongst a source and a target ontology. Haase et al. (2008) devised the NeON Toolkit, which is similar in spirit to the Protégé tool, but also includes a module to define correspondences amongst a source and a target ontology.

2.3. *Discussion*

After surveying current tools to exchange data, we conclude that some of them were designed for nested-relational data models and are thus not applicable to ontologies. Neither seem they easy to adapt due to a number of inherent differences (Motik et al., 2009; Rivero et al., 2011a), namely:

- A nested-relational data model defines a tree that comprises a number of nodes, which may be nested and have an arbitrary number of attributes, and it is also possible to specify referential constraints to relate attributes; contrarily, an ontology is not a tree, but a graph in which there is not a root node, and it can contain cycles.

- A nested-relational data model does not support the specialisation of nodes; contrarily, an ontology allows to specialise classes and properties in taxonomies. Furthermore, these taxonomies may be completely arbitrary in the sense that all possible relations are allowed, such as an undefined number of specialisation levels, a class (property) may have more than one superclass (superproperty), or a class (property) is subclass (subproperty) of itself.

- An instance in a nested-relational data model has a unique type that corresponds to an existing node; contrarily, an instance in an ontology may have multiple types of several existing classes.

- In nested-relational data models, queries to exchange data are encoded using XQuery or XSLT, which build on the structure of the XML documents on which they are executed; contrarily, in an ontology, queries must be encoded in a language that is independent from the structure of the documents used to represent it, e.g., XML, N3, or Turtle.

Furthermore, the tools that are specifically tailored towards ontologies suffer from a number of problems, namely: a) Maedche et al. (2002), and Mocan and Cimpian (2007) make no attempt to identify groups of correspondences that must be taken into account together, which may easily lead to incoherent target data, i.e., data that does not satisfy the constraints in the target ontology (Bernstein and Melnik, 2007; Popa et al., 2002); b) Bizer and Schultz (2010), Dou et al. (2005), Parreiras et al. (2008), and Ressler et al. (2007) do not build on correspondences, but require users to handcraft and maintain their executable mappings using several languages; c) Gennari et al. (2003), Haase et al. (2008), and Noy and Musen (2003) do not focus on the exchange of data, but on the definition or automatic generation of correspondences; they do not provide any mechanisms to transform these correspondences into executable mappings.

3. Preliminaries

In this article, we focus on ontologies that are modelled using the OWL 2 Lite profile ontology language. Furthermore, we also focus on exchanging data using SPARQL 1.1 query

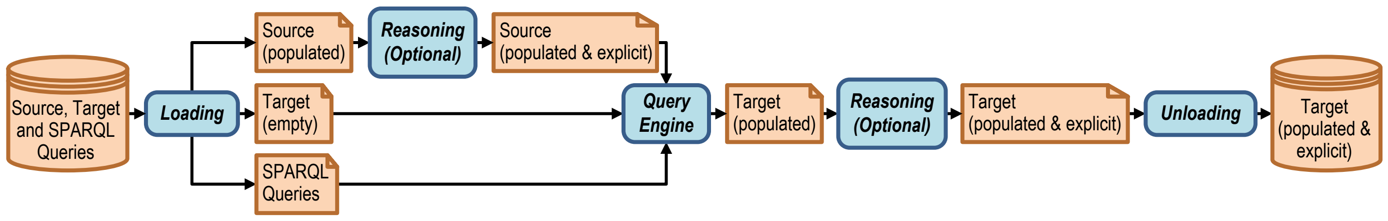


Figure 1: Steps to exchange data by means of a query engine.

engines. In the rest of this section, we present some preliminaries on ontologies and exchanging data amongst them in Sections 3.1 and 3.2, respectively. Finally, we present our running example in Section 3.3.

3.1. *Semantic-web ontologies*

An OWL ontology comprises a set of entities that are identified by URIs. An entity may be a class, a data property, or an object property. A class can be specialised into other classes. Data properties have a set of classes as domain and a basic XSD data type as range. Object properties have a set of classes as domain and range. Beyond class, subclass, property, domain, and range constructs, the OWL 2 Lite profile ontology language provides other constructs to represent other constraints, e.g., *rdfs*:*subPropertyOf*, which allows to model subproperty constraints; *owl*:*sameAs*, which deals with relating two instances that model the same real-world object; *owl*:*minCardinality*, which restricts the minimal number of property instances that an ontology should contain; or *owl*:*versionInfo*, which is devised to provide meta-information.

In addition to structure, ontologies also provide data: a class instance is identified by its own URI, and it may have a number of types. A data property instance relates a class instance with a literal by means of a data property. An object property instance relates two class instances. RDF, which is based on triples, is used to represent both the structure and data of an ontology by means of an object property. A triple comprises three elements: the first one is called subject, the second one is called predicate, and the third one is called object. Triples are used to represent both the structure and the data of an ontology.

3.2. *Exchanging data*

Exchanging data amongst ontologies by means of a SPARQL query engine comprises five steps (see Figure 1), namely:

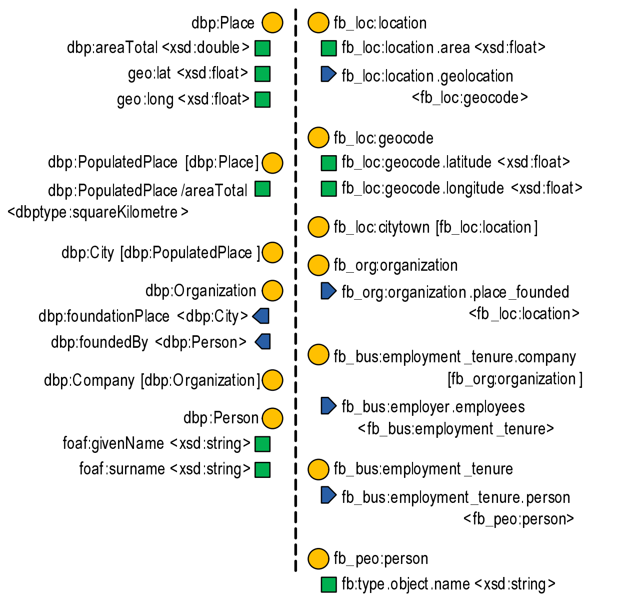
1. Loading: This step consists of loading the source and target ontologies and the set of SPARQL queries from a persistent storage into the appropriate internal data structures.
2. Reasoning over source: This step is optional and deals with making it explicit the knowledge in the source ontology (Mokhtar et al., 2008). Note that SPARQL 1.1 engines implement RDF Schema and OWL entailments, which would make this step unnecessary. However, current query engines are far from fully supporting SPARQL 1.1 entailments (SPARQL, 2012), which motivated us not to rely on this feature of SPARQL 1.1.
3. Query engine: This step consists of executing a set of SPARQL queries over the source ontology to produce instances of the target ontology. The result of this step must be the same regardless of the order in which queries are executed.
4. Reasoning over target: This step is also optional and it deals with making it explicit the knowledge in the target ontology.
5. Unloading: This step deals with saving the target ontology from the internal data structures to a persistent storage.

SPARQL queries are used to retrieve and construct triples from RDF stores. SPARQL provides four types of queries, namely: SELECT, CONSTRUCT, ASK, and DESCRIBE. They are based on triple patterns that are similar to triples, but allow to specify variables in the subject, predicate, and/or object, which are prefixed with a '?'. In this article, we focus on the CONSTRUCT type, since this type of queries allows to both retrieve and construct RDF data. Finally, BIND clauses are used to implement transformation functions.

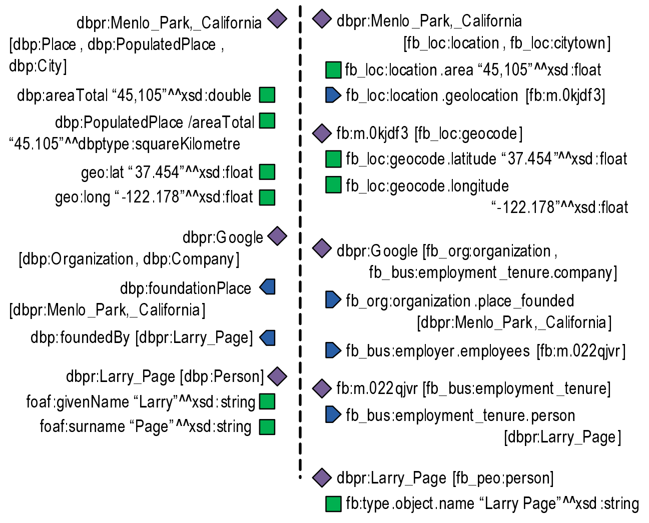
3.3. *Running example*

To illustrate the exchanging of data amongst ontologies, we provide a running example that consists of exchanging data from a part of DBpedia to a part of Freebase (see Figure 2). On the one hand, DBpedia (Bizer et al., 2009) is a community effort to annotate and make the data stored at Wikipedia accessible by means of an ontology. On the other hand, Freebase (Bollacker et al., 2008) is another community effort to store structured data in an open repository that is exposed using RDF. Figure 2(a) shows the structure of both DBpedia (left side) and Freebase (right side) ontologies using a tree notation. These ontologies model companies, the cities where they were founded, and the people who founded them.

In the figure, *dbp*:*Place* is a class that models a geographic place, and we denote it as a circle. (Throughout this article, we use a number of namespaces as prefixes that are summarised in Table 2.) *dbp*:*PopulatedPlace* is a subclass of *dbp*:*Place*, and we denote it as *dbp*:*PopulatedPlace* [*dbp*:*Place*]. An example of a data property is *geo*:*lat*, which models the latitude of a geographic position, and we denote it as a square. The domain of *geo*:*lat* is {*dbp*:*Place*}, and we denote it by nesting *geo*:*lat* into *dbp*:*Place*. The range of *geo*:*lat* is *xsd*:*float*, and we denote it as *geo*:*lat* <*xsd*:*float*>. An example of an object property is *dbp*:*foundationPlace*, which models the place in which an organisation was founded, and we denote it as a pentagon. The


dbp:Place
dbp:areaTotal <xsd:double>
geo:lat <xsd:float>
geo:long <xsd:float>
dbp:PopulatedPlace [dbp:Place]
dbp:PopulatedPlace /areaTotal <dbptype:squareKilometre>
dbp:City [dbp:PopulatedPlace]
dbp:Organization
dbp:foundationPlace <dbp:City>
dbp:foundedBy <dbp:Person>
dbp:Company [dbp:Organization]
dbp:Person
foaf:givenName <xsd:string>
foaf:surname <xsd:string>
fb_loc:location
fb_loc:location.area <xsd:float>
fb_loc:location.geolocation <fb_loc:geocode>
fb_loc:geocode
fb_loc:geocode.latitude <xsd:float>
fb_loc:geocode.longitude <xsd:float>
fb_loc:citytown [fb_loc:location]
fb_org:organization
fb_org:organization.place_founded <fb_loc:location>
fb_bus:employment_tenure.company [fb_org:organization]
fb_bus:employer.employees <fb_bus:employment_tenure>
fb_bus:employment_tenure
fb_bus:employment_tenure.person <fb_peo:person>
fb_peo:person
fb:type.object.name <xsd:string>

(a) Structure of the sample ontologies.


dbpr:Menlo_Park,_California [dbp:Place, dbp:PopulatedPlace, dbp:City]
dbp:areaTotal "45,105"^^xsd:double
dbp:PopulatedPlace /areaTotal "45.105"^^dbptype:squareKilometre
geo:lat "37.454"^^xsd:float
geo:long "-122.178"^^xsd:float
dbpr:Google [dbp:Organization, dbp:Company]
dbp:foundationPlace [dbpr:Menlo_Park,_California]
dbp:foundedBy [dbpr:Larry_Page]
dbpr:Larry_Page [dbp:Person]
foaf:givenName "Larry"^^xsd:string
foaf:surname "Page"^^xsd:string
dbpr:Menlo_Park,_California [fb_loc:location, fb_loc:citytown]
fb_loc:location.area "45,105"^^xsd:float
fb_loc:location.geolocation [fb:m.0kjdf3]
fb:m.0kjdf3 [fb_loc:geocode]
fb_loc:geocode.latitude "37.454"^^xsd:float
fb_loc:geocode.longitude "-122.178"^^xsd:float
dbpr:Google [fb_org:organization, fb_bus:employment_tenure.company]
fb_org:organization.place_founded [dbpr:Menlo_Park,_California]
fb_bus:employer.employees [fb:m.022qjvr]
fb:m.022qjvr [fb_bus:employment_tenure]
fb_bus:employment_tenure.person [dbpr:Larry_Page]
dbpr:Larry_Page [fb_peo:person]
fb:type.object.name "Larry Page"^^xsd:string

(b) Data of the sample ontologies.

```
q1: CONSTRUCT{
  ?l  rdf:type   fb_loc:location .
}WHERE {
  ?l  rdf:type   dbp:Place . }

q2: CONSTRUCT{
  ?l  rdf:type   fb_loc:location .
  ?l  fb_loc:location.area ?a .
}WHERE {
  ?l  rdf:type    dbp:Place .
  ?l  dbp:areaTotal   ?at .
  BIND(dbToFl(?at) AS ?a) }

q3: CONSTRUCT{
  ?c   rdf:type   fb_org:organization .
  ?c   rdf:type   fb_bus:employment_tenure.company .
  ?p   rdf:type   fb_peo:person .
  _:e  rdf:type   fb_bus:employment_tenure .
  ?c   fb_bus:employer.employees   _:e .
  _:e  fb_bus:employment_tenure.person  ?p .
}WHERE {
  ?c rdf:type   dbp:Organization .
  ?c rdf:type   dbp:Company .
  ?p rdf:type   dbp:Person .
  ?c dbp:foundedBy   ?p . }

q4: CONSTRUCT{
  ?p rdf:type   fb_peo:person .
  ?p fb:type.object.name ?n .
}WHERE {
  ?p rdf:type   dbp:Person .
  ?p foaf:givenName  ?gn .
  ?p foaf:surname  ?sn .
  BIND(concat(?gn, ?sn) AS ?n) }
```

(c) Sample queries to exchange data.

Figure 2: Running example.

Prefix	URI
dbp	`http://dbpedia.org/ontology/`
dbpr	`http://dbpedia.org/resource/`
dbptype	`http://dbpedia.org/datatype/`
fb	`http://rdf.freebase.com/ns/`
fb_loc	`http://rdf.freebase.com/ns/location.`
fb_peo	`http://rdf.freebase.com/ns/people.`
fb_bus	`http://rdf.freebase.com/ns/business.`
fb_org	`http://rdf.freebase.com/ns/organization.`
foaf	`http://xmlns.com/foaf/0.1/`
geo	`http://www.w3.org/2003/01/geo/wgs84_pos#`
owl	`http://www.w3.org/2002/07/owl#`
rdf	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
rdfs	`http://www.w3.org/2000/01/rdf-schema#`
xsd	`http://www.w3.org/2001/XMLSchema#`

Table 2: Prefixes used throughout the article.

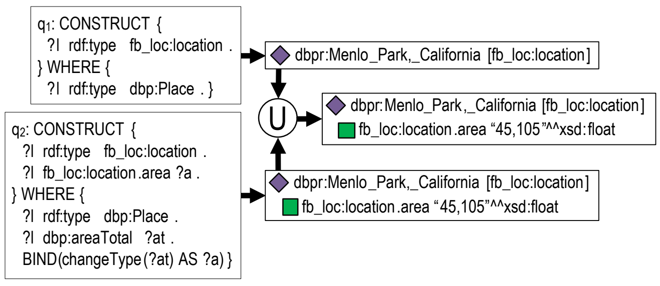
domain of *dbp:foundationPlace* is {*dbp:Organization*} and the range is {*dbp:City*}.

Regarding the ontology data (see Figure 2(b)), we denote a class instance as a diamond, e.g., *dbpr:Menlo_Park,_California* is a class instance of class *dbp:City*. *dbpr:Menlo_Park,_California* is also an instance of types {*dbp:Place*, *dbp:PopulatedPlace*}, since *dbp:City* is a subclass of *dbp:PopulatedPlace*, and *dbp:PopulatedPlace* is subclass of *dbp:Place*; reasoners are used to make this knowledge explicit. We denote each data property instance as a square, e.g., *geo:lat* relates *dbpr:Menlo_Park,_California* with "37.454"^^*xsd:float*. Furthermore, we denote each object property instance as a pentagon, e.g., *dbp:foundedBy* relates *dbpr:Google* and *dbpr:Menlo_Park,_California*.

Some sample RDF triples in our example are the following:

(*dbp:Organization*,	*rdf:type*,	*owl:Class*)
(*dbp:Company*,	*rdf:type*,	*owl:Class*)
(*dbp:Company*,	*rdfs:subClassOf*,	*dbp:Organization*)
(*dbp:Person*,	*rdf:type*,	*owl:Class*)
(*dbpr:Google*,	*rdf:type*,	*dbp:Organization*)
(*dbpr:Google*,	*rdf:type*,	*dbp:Company*)
(*dbpr:Larry_Page*,	*rdf:type*,	*dbp:Person*)
(*dbpr:Google*,	*dbp:foundedBy*,	*dbpr:Larry_Page*)

Figure 2(c) shows four examples of SPARQL queries of the CONSTRUCT type that are expected to exchange data in our running example: q_1 retrieves instances of *dbp:Place* (the WHERE clause) and reclassifies them as *fb_loc:location* (the CONSTRUCT clause). q_2 is similar to q_1 but also transforms the value of *dbp:areaTotal* into *fb_loc:location.area* by means of function *dbToFl*, which transforms a double into a float. Note that this requires to use a BIND clause in which the result of applying the function is assigned to variable ?*a*. q_3 retrieves a company with its founder (property *dbp:foundedBy*) and stores



```
q1: CONSTRUCT {
  ?l rdf:type   fb_loc:location .
} WHERE {
  ?l rdf:type   dbp:Place . }
```

```
q2: CONSTRUCT {
  ?l rdf:type   fb_loc:location .
  ?l fb_loc:location.area ?a .
} WHERE {
  ?l rdf:type   dbp:Place .
  ?l dbp:areaTotal   ?at .
  BIND(changeType(?at) AS ?a) }
```

◆ dbpr:Menlo_Park,_California [fb_loc:location]
U
◆ dbpr:Menlo_Park,_California [fb_loc:location]
🟩 fb_loc:location.area "45,105"^^xsd:float
◆ dbpr:Menlo_Park,_California [fb_loc:location]
🟩 fb_loc:location.area "45,105"^^xsd:float

Figure 3: Union of two sample overlapping queries.

them at the target, which models the same information with a different structure, i.e., it relates the company and the founder by means of a new instance of type *fb_bus*:*employment_tenure*; since this type is not present in the source, we use a blank node that acts as a placeholder for data that is not available when the data exchange is performed. Blank nodes are prefixed with '_:', and they are known as labelled nulls in the context of nested-relational data models (Fagin et al., 2005). Finally, q_4 retrieves the first and last name of a person and generates the full name in the target by means of function *concat*.

Note that queries q_1 and q_2 overlap; however, they both are necessary to perform data exchange since there may be instances of type *dbp*:*Place* that are not related to any instances of property *dbp*:*areaTotal*; these instances would not be exchanged by q_2 in isolation, and we have to consider every possible combination of data. However, this overlapping does not produce any incoherent data. For example, Figure 3 presents some target sample data that have been constructed using q_1 and q_2; even if each result in isolation comprises an instance of *dbpr*:*Menlo_Park*, *_California*, the union of the results of both queries comprises only one class instance of *dbpr*:*Menlo_Park*, *_California*, which is the expected result.

4. Generating executable mappings

In this section, we describe the core of MostoDE, in which we present the underlying data model of our tool (see Section 4.1), and the algorithms to automatically generate SPARQL executable mappings, which are divided into two groups: the generation of kernels (see Section 4.2) and the transformation of kernels into executable mappings in SPARQL 1.1 (see Section 4.3).

4.1. The underlying data model

This model defines the concepts that we need to automatically generate executable mappings in SPARQL (see Figure 4). The main concept of this data model is *Entity*, which represents the entities that form an ontology. An *Entity* has a URI and specialises into either *Class*, *DataProperty*, or *ObjectProperty*. Furthermore, ontologies comprise a number of constraints that restrict how these entities should be combined; we use concept *Constraint* to model them; *Constraint* specialises into either *Domain*, *Range*, *StrongDomain*, *StrongRange*, *Subclass*,

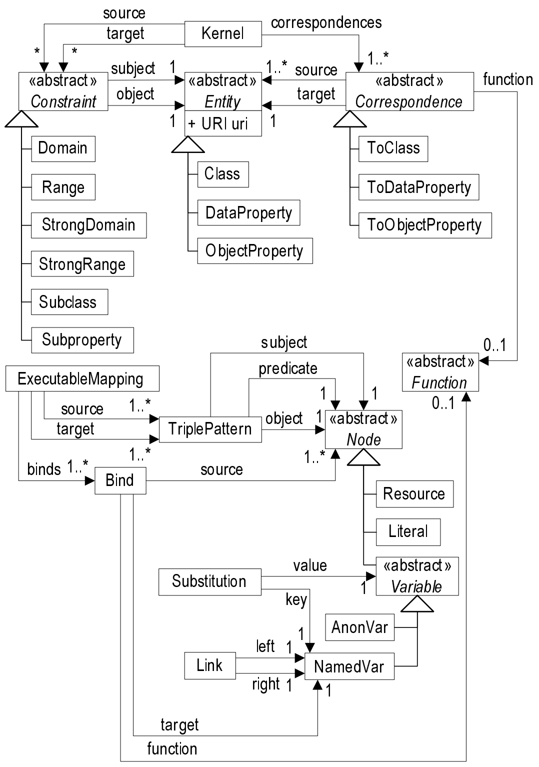


Figure 4: Modelling entities, constraints, correspondences, kernels, links, substitutions, and executable mappings.

or *Subproperty*. A *Constraint* relates two instances of *Entity* by means of relations *subject* and *object*.

Correspondence is the concept that represents a correspondence, i.e., a hint that relates some source entities to a target entity; a correspondence relates one or more instances of *Entity* by means of the *source* relation, with a single instance of *Entity* by means of the *target* relation. Additionally, it is possible to define transformation functions (*Function*) from some source entities onto a target entity. It is important to notice that these functions can be used to perform simple data transformations like concatenating a name and a surname to form a full name, or to invoke external functionality like looking up a code in a database to dereference it.

Correspondence specialises into *ToClass*, *ToDataProperty*, or *ToObjectProperty*. A *Kernel* relates two sets of instances of *Constraint* by means of relations *source* and *target*, and a set of instances of *Correspondence* using the *correspondences* relation. Relations *source* and *target* are have a minimal cardinality of zero, which entails that a *Kernel* may not be related to any instances of *Constraint*; however, a *Kernel* must be related to, at least, an instance of *Correspondence*.

ExecutableMapping represents a SPARQL query of the CONSTRUCT type, which means that it comprises a CONSTRUCT clause, a WHERE clause, and a number of BIND clauses, which are represented by means of relations *source*, *target*, and *binds*, respectively. Relations *source* and *target* represent sets of *TriplePatterns*. Each *TriplePattern* is related to three instances of *Node* by means of relations *subject*,

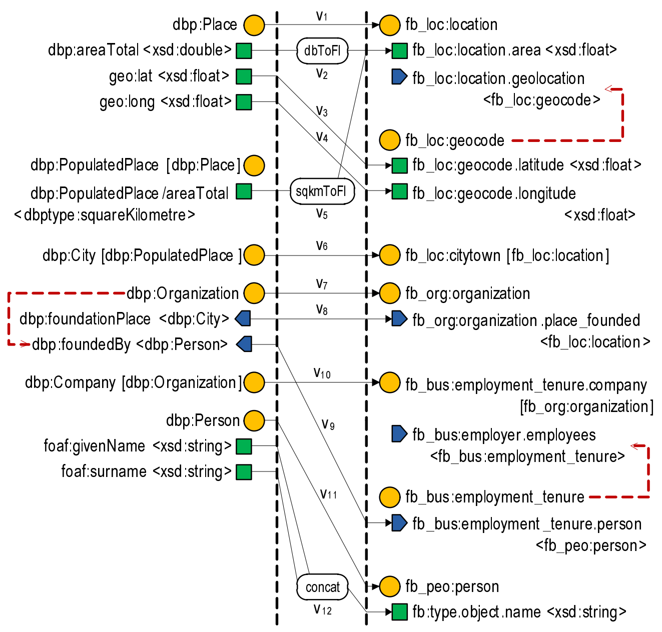

dbp:Place
dbp:areaTotal <xsd:double>
geo:lat <xsd:float>
geo:long <xsd:float>
dbp:PopulatedPlace [dbp:Place]
dbp:PopulatedPlace /areaTotal <dbptype:squareKilometre>
dbp:City [dbp:PopulatedPlace]
dbp:Organization
dbp:foundationPlace <dbp:City>
dbp:foundedBy <dbp:Person>
dbp:Company [dbp:Organization]
dbp:Person
foaf:givenName <xsd:string>
foaf:surname <xsd:string>
v1
dbToFl
v2
v3
v4
sqkmToFl
v5
v6
v7
v8
v10
v9
v11
concat
v12
fb_loc:location
fb_loc:location.area <xsd:float>
fb_loc:location.geolocation <fb_loc:geocode>
fb_loc:geocode
fb_loc:geocode.latitude <xsd:float>
fb_loc:geocode.longitude <xsd:float>
fb_loc:citytown [fb_loc:location]
fb_org:organization
fb_org:organization.place_founded <fb_loc:location>
fb_bus:employment_tenure.company [fb_org:organization]
fb_bus:employer.employees <fb_bus:employment_tenure>
fb_bus:employment_tenure
fb_bus:employment_tenure.person <fb_peo:person>
fb_peo:person
fb:type.object.name <xsd:string>

Figure 5: Structure of the ontologies, constraints and correspondences in our running example.

predicate, and *object*. A *Node* specialises into either a *Resource*, a *Literal*, or a *Variable*, which also specialises into either an *AnonVar* (a blank node) or a *NamedVar*. A *Bind* relates a number of instances of *Node* and a *NamedVar* by means of *source* and *target* relations, respectively. Furthermore, a *Bind* may be optionally related to an instance of *Function* using the *function* relation.

We also use two auxiliary concepts: *Link* and *Substitution*. A *Link* relates two instances of *NamedVar* using the *left* and *right* relations, which are used to link free named variables when transforming a kernel into a SPARQL executable mapping (see Section 4.3). A *Substitution* represents a map, in which the key is an instance of *NamedVar* (the *key* relation), and the value is an instance of *Variable* (the *value* relation). Substitutions are used to rename free named variables that are linked to the same named or anonymous variable (see Section 4.3).

Our tool allows to define six types of constraints and three types of correspondences, which are illustrated in Figure 5. There, we present the source and target ontologies of our running example, and the constraints and correspondences that are needed to exchange data from the source into the target ontology by means of the automatically generated SPARQL executable mappings. The constraints that our tool supports are the following:

- *Domain*: this constraint relates an instance of *DataProperty* or *ObjectProperty* (*subject*) with an instance of *Class* (*object*). For example, in Figure 5, the domain of data property *dbp*:*areaTotal* is class *dbp*:*Place*, or the domain of object property *dbp*:*foundationPlace* is class *dbp*:*Organization*.

- *Range*: it relates an instance of *ObjectProperty* (*subject*) with an instance of *Class* (*object*). For example, in Figure 5, the range of object property *dbp*:*foundationPlace* is class *dbp*:*City*.

- *StrongDomain*: it relates an instance of *Class* (*subject*) with an instance of *DataProperty* or *ObjectProperty* (*object*). Note that this constraint entails that the class has a minimal cardinality of one with respect to the data or object property. For example, in Figure 5, the strong domain of class *dbp*:*Organization* is object property *dbp*:*foundedBy*, and we denote it as a dashed arrow from the class to the property.

- *StrongRange*: it relates an instance of *Class* (*subject*) with an instance of *ObjectProperty* (*object*). Note that this constraint entails that the class has a minimal cardinality of one with respect to the object property. For example, in Figure 5, the strong range of class *fb_loc*:*geocode* is object property *fb_loc*:*location.geolocation*, and we denote it as a dashed arrow from the class to the property.

- *Subclass*: it relates two instances of *Class* (*subject* and *object*, respectively). For example, in Figure 5, *dbp*:*PopulatedPlace* is a subclass of *dbp*:*Place*.

- *Subproperty*: it relates two instances of *DataProperty* or *ObjectProperty* (*subject* and *object*, respectively). Note that, in the running example of Figure 5, we have no example of subproperty constraints. (It is not so common to find subproperties in the Web of Data (Glimm et al., 2012).)

In addition to these constraints, our tool allows to define a number of correspondences amongst source and target ontologies, which are the following:

- *ToClass*: this correspondence relates one or more instances of *Entity* (*source* relation) that are reclassified into a single instance of *Class* (*target* relation). When there are more than one source entities, it is mandatory to use a transformation function to reclassify the instances. For example, in Figure 5, v_1 is a to-class correspondence that relates a single instance of class *dbp*:*Place* with another single instance of class *fb_loc*:*location*.

- *ToDataProperty*: this correspondence relates one or more instances of *Entity* (*source* relation) that are copied to a single instance of *DataProperty* (*target* relation). If there are multiple source entities, then it is mandatory to use a transformation function. For example, in Figure 5, v_2 is a to-data-property correspondence that relates a single instance of *dbp*:*areaTotal* that is copied to a single instance of *fb_loc*:*location.area* data property by means of the *dbToFl* function. Another example is v_3, which relates *geo*:*lat* with *fb_loc*:*geocode.latitude* using the implicit identity function. Correspondence v_{12} relates two source data properties, *foaf*:*givenName* and *foaf*:*surname*, with a target data property *fb*:*type.object.name* using the *concat* function.

```
 1: $GenerateExecutableMappings$
 2: Input
 3:     $C_S, C_T$: Set of Constraint
 4:     $V$: Set of Correspondence
 5: Output
 6:     $M$: Set of ExecutableMapping
 7: Variables
 8:     $k$: Kernel
 9:     $m$: ExecutableMapping
10:     $v$: Correspondence
11:
12: $M = \emptyset$
13: For each $v \in V$
14:     $k = GenererateKernel(v, C_S, C_T, V)$
15:     $m = TransformIntoExecutableMapping(k)$
16:     $M = M \cup \{m\}$
```

Figure 6: Algorithm to generate executable mappings.

- *ToObjectProperty*: this correspondence relates one or more instances of *Entity* (*source* relation) to a single instance of *ObjectProperty* (*target* relation) that must exist at least in the target. This correspondence does not reclassify or copy source instances to target instances, but it only specifies that a target instance should exist only if there exist a number of source instances. For example, in Figure 5, v_8 is a to-object-property correspondence that relates *dbp*:*foundationPlace* to *fb_org*:*organization.place_founded*, i.e., an instance of *fb_org*:*organization.place_founded* in the target exists only if there exists, at least, one instance of *dbp*:*foundationPlace* in the source. (It is not so common to find to-object-property correspondences with more than one object property in the Web of Data (Bizer and Schultz, 2010).)

It is important to notice that these definitions of correspondences are inherently incomplete, i.e., a correspondence of type *ToDataProperty* states how to generate the object of an instance of a given target data property, but it does not state how to generate the subject of this instance; in addition, a correspondence of type *ToObjectProperty* does not state how to generate the subject or the object of an instance of a given target object property; it only states its existence. Therefore, it is mandatory to combine the correspondences to generate executable mappings.

Figure 6 shows our algorithm to generate SPARQL executable mappings. This algorithm takes a set of source and target constraints, and a set of correspondences as input, and it outputs a number of executable mappings in SPARQL: for each correspondence (line 13), we first generate a kernel, which is a subproblem of the input data exchange problem that describes source data that needs to be exchanged as a whole, and target data that needs to be created as a whole (line 14), and then transform it into a SPARQL executable mapping (line 15). An important issue is that both constraints and correspondences relate source and target entities, which is the reason why it is possible to group correspondences by means of constraints. In the following subsections, we explain how we automatically generate kernels and transform them into executable mappings.

```
 1: $GenerateKernel$
 2: Input
 3:     $v$: Correspondence
 4:     $C_S, C_T$: Set of Constraint
 5:     $V$: Set of Correspondence
 6: Output
 7:     $k$: Kernel
 8: Variables
 9:     $v'$: Correspondence
10:     $E_S, E_T$: Set of Entity
11:     $V'$: Set of Correspondence
12:
13: - Compute source and target constraints
14: $k.source$ = expand $v.source$ in $C_S$
15: $k.target$ = expand $\{v.target\}$ in $C_T$
16: - Find correspondences
17: $E_S$ = get entities from $k.source$
18: $E_T$ = get entities from $k.target$
19: $V' = \emptyset$
20: For each $v' \in V$
21:     If $v'.source \subseteq E_S \land v.target \in E_T$
22:         $V' = V' \cup \{v'\}$
23: $k.correspondences = V'$
```

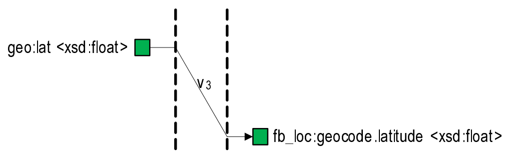
Figure 7: Algorithm to generate kernels.

4.2. Generating kernels

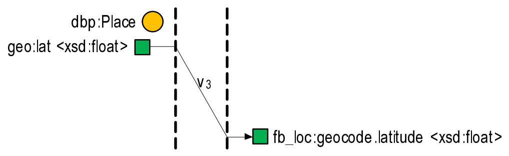
For each correspondence in the given input, our tool automatically generates a kernel. Intuitively, a kernel comprises a subset of source entities and constraints, target entities and constraints, and correspondences that describe a subset of data that requires to be exchanged as a whole, i.e., if more or less data are considered, then the exchange would be incoherent. Figure 7 presents the algorithm to automatically generate them, which takes a single correspondence v, a set of source and target constraints, and a set of correspondences as input, and it outputs the kernel that is derived from correspondence v.

First, this algorithm takes source and target entities of the input correspondence v and expands them using the constraints (lines 14 and 15 in Figure 7); note that *v.source* is a set of entities, whereas *v.target* is a single entity since we deal with n:1 correspondences. This expansion consists of finding the source or target constraints that result for exploring the source or target entities in depth using the constraints. For example, Figure 8(a) shows correspondence v_3 of our running example (see Figure 5), whose source entity is *geo*:*lat*; thus expanding this entity by means of the source constraints results in the partial kernel of Figure 8(b). The target entity of v_3 is *fb_loc*:*geocode.latitude*, thus expanding this entity by means of the target constraints results in the partial kernel of Figure 8(c).

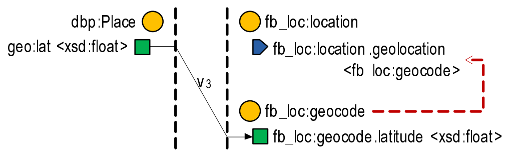
Finally, we have to cluster other correspondences that may be implied in this kernel; to perform this, we add a correspondence v to the kernel if the source entities of v are included in the entities of the source constraints, and the target entity of v is included in the entities of the target constraints (lines 17–23 in Figure 7). Figure 8(d) shows the correspondences that relate the



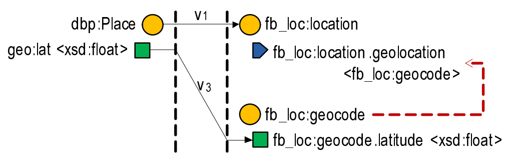
(a) Correspondence v_3 of our running example.



(b) Expansion of the source entities of v_3.



(c) Expansion of the target entities of v_3.



(d) Finding correspondences of the kernel.

Figure 8: Example of the automatic kernel generation.

entities of the previous example, in this case, v_1, which relates *dbp*:*Place* with *fb_loc*:*location*.

4.3. Transforming kernels into executable mappings

Figure 9 presents the algorithm that takes a kernel as input and outputs an executable mapping. First, we generate two sets of triple patterns and a set of BIND clauses using the constraints and the correspondences of the input kernel (lines 22 and 24). When initialising triple patterns, we create a single triple pattern for each entity that belongs to either the source or target constraints; note that triple patterns created from the source (target) constraints form the WHERE (CONSTRUCT) clause. If the entity is a class, we generate a triple pattern of the form: $?x$ *rdf*:*type* C, where $?x$ is a fresh variable and C is the entity; if the entity is a data or object property, we generate a triple of the form: $?x\ P\ ?y$, where $?x$ and $?y$ are fresh variables and P is the entity. Furthermore, for each to-class or to-data-property correspondences, we generate a single BIND clause of the form: *BIND*($f(?x_1, ?x_2, \ldots, ?x_n)$ *AS* $?y$), where $?x_1$, $?x_2$, …, $?x_n$ and $?y$ are fresh variables and f is the corresponding transformation function; note that we do not generate any BIND clauses for to-object-property correspondences since they specify that a target instance should exist only if there exist a number of source instances, but do not reclassify or copy source instances to target instances. Figure 10(a) shows the executable mapping that results after this initialisation for the kernel depicted in Figure 8(d).

After the initialisation of triple patterns and BIND clauses, we compute a set of links amongst their fresh variables by

```
 1: $TransformIntoExecutableMapping$
 2: Input
 3:     $k$: Kernel
 4: Output
 5:     $m$: ExecutableMapping
 6: Variables
 7:     $C_S, C_T$: Set of Constraint
 8:     $V$: Set of Correspondence
 9:     $E_S, E_T$: Set of Entity
10:     $T_C, T_W$: Sorted Set of TriplePattern
11:     $B$: Set of Bind
12:     $L_C, L_W, L_B$: Set of Link
13:     $S$: Set of Substitution
14:
15: - Initialise auxiliar variables
16: $C_S$ = $k.source$
17: $C_T$ = $k.target$
18: $V$ = $k.correspondences$
19: $E_S$ = get entities from $C_S$
20: $E_T$ = get entities from $C_T$
21: - Initialise WHERE, CONSTRUCT and BIND
22: $T_W$ = initialise patterns using $E_S$
23: $T_C$ = initialise patterns using $E_T$
24: $B$ = initialise binds using $V$
25: - Compute constraint and bind links
26: $L_W$ = compute links in $T_W$ using $C_S$
27: $L_C$ = compute links in $T_C$ using $C_T$
28: $L_B$ = compute links in $B$ using $V$
29: - Compute substitution
30: $S$ = compute substitution using $L_W \cup L_C \cup L_B$
31: - Apply substitution
32: $T_W$ = apply $S$ to $T_W$
33: $T_C$ = apply $S$ to $T_C$
34: $B$ = apply $S$ to $B$
35: - Create executable mapping
36: $m.source$ = sort $T_W$ using heuristics
37: $m.target$ = sort $T_C$ using heuristics
38: $m.binds$ = $B$
```

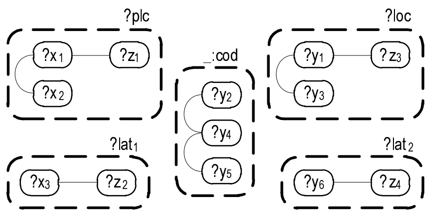
Figure 9: Algorithm to transform kernels into executable mappings.

analysing the constraints and the correspondences in the kernel that is associated with the correspondence being analysed (lines 26 and 28). In this case, we create a link between two variables if there is a constraint or a correspondence that relates them. For example, Figure 10(b) presents the links between the variables of the initial executable mapping (see Figure 10(a)), in which $?x_1$ and $?x_2$ are linked because the domain of *geo*:*lat* is *dbp*:*Place*, $?x_1$ and $?z_1$ are linked because v_1 has *dbp*:*Place* as the source, $?y_2$ and $?y_4$ are linked because the range of *fb_loc*:*location.geolocation* is *fb_loc*:*geocode*, or $?y_1$ and $?z_3$ are linked because v_1 has *fb_loc*:*location* as the target.

Then, we compute a substitution for every fresh variable using the previously defined links (line 30). To perform this, it is important to notice that the previous result is a graph in which every connected component includes a group of variables that are linked and should actually be the same. Therefore, we transform this graph into a substitution in which every variable

```
m1: CONSTRUCT{
  ?y1  rdf:type  fb_loc:location .
  ?y2  rdf:type  fb_loc:geocode .
  ?y3  fb_loc:location.geolocation  ?y4 .
  ?y5  fb_loc:geocode.latitude      ?y6 .
}WHERE {
  ?x1  rdf:type  dbp:Place .
  ?x2  geo:lat   ?x3 .
  BIND (?z1 AS ?z3)
  BIND (?z2 AS ?z4)}
```

(a) Initial executable mapping.



(b) Links and substitution.

```
m1: CONSTRUCT {
  ?loc  fb_loc:location.geolocation  _:cod .
  _:cod fb_loc:geocode.latitude      ?lat2 .
  ?loc  rdf:type  fb_loc:location .
  _:cod rdf:type  fb_loc:geocode .
}WHERE {
  ?plc  geo:lat   ?lat1 .
  ?plc  rdf:type  dbp:Place .
  BIND (?plc AS ?loc)
  BIND (?lat1 AS ?lat2)}
```

(c) Final executable mapping.

Figure 10: Example of the automatic transformation of a kernel into an SPARQL executable mapping.

in every connected component is replaced by the same fresh variable. Figure 10(b) shows these substitutions, in which $?x_1$, $?x_2$, and $?z_1$ are replaced by $?plc$; $?x_3$, and $?z_2$ are replaced by $?lat_1$; $?y_1$, $?y_3$, and $?z_3$ are replaced by $?loc$; $?y_6$ and $?z_4$ are replaced by $?lat_2$; and $?y_2$, $?y_4$, and $?y_5$ are replaced by $\_:cod$. In the last case, there exist a number of variables in the CONSTRUCT clause that are not linked to any variable in the BIND clauses; this entails that the set of correspondences is not complete enough to describe the data exchange problem that is being analysed. In some situations, the set of correspondences can be completed to solve the problem, but there are others, as in our running example, in which this is not possible because the target ontology provides more information than the source ontology. Therefore, instead of failing to exchange any data due to this problem, we generate a blank node that acts as a placeholder.

In the next step, we deal with replacing each variable of the CONSTRUCT, WHERE and BIND clauses that belongs to the same connected component by the same fresh variable using the previously computed substitution (lines 32 and 34).

Last, but not least, we have found out that the ARQ SPARQL engine that we used in our implementation did not optimise query execution, which motivated us to work on a number of heuristics that helped reduce the execution times from days to seconds. Note that both T_C and T_W are sorted sets in the algorithm in Figure 9; the heuristics we use to sort them are the following:

1. Triple patterns that have a property as predicate are at the top of a sorted set, and triple patterns that have *rdf:type* as predicate are at the bottom. The intuition behind this heuristic is that we should not retrieve the cartesian product of every instance of a given class, and then prune the instances using the properties related to them, but we should retrieve only those instances that are related by the given properties.
2. Let t_1 and t_2 be two triple patterns, each of which has a property as predicate, t_1 is already in a sorted set, and we wish to include t_2. We can distinguish the following cases: 1) if $t_1.subject$ is equal to $t_2.subject$ or $t_2.object$, then add t_2 before t_1; 2) if $t_1.object$ is equal to $t_2.subject$ or $t_2.object$, then add t_2 after t_1; 3) otherwise, add t_2 after t_1. The intuition behind this heuristic is that we prune more instances if the triple patterns are concatenated; otherwise, we retrieve the cartesian product of instances.

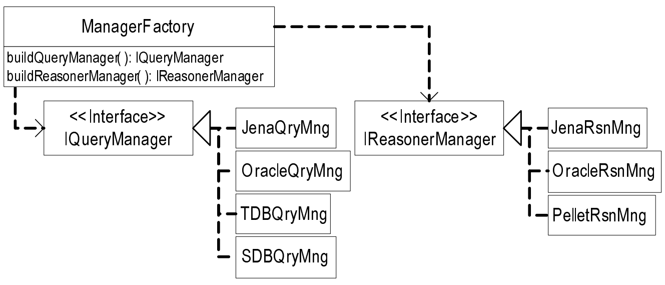


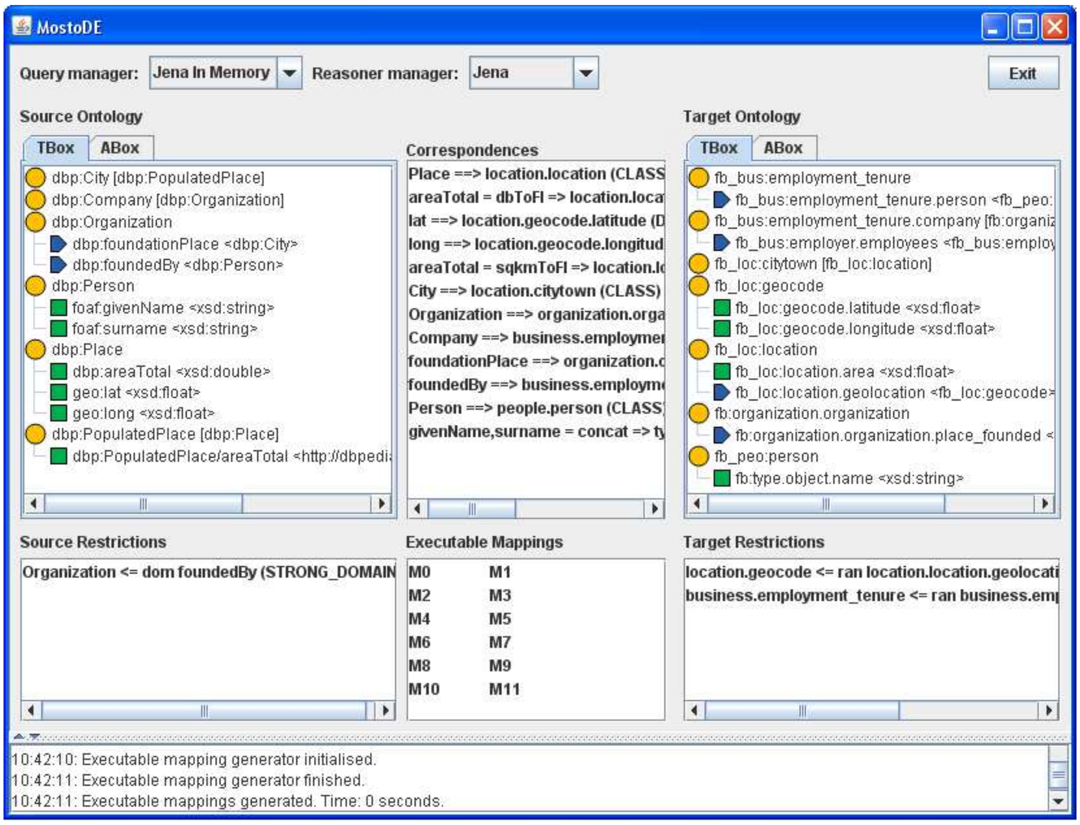
Figure 11: Design diagram of query and reasoner managers.

Taking these heuristics into account, Figure 10(c) shows the final SPARQL executable mapping that has been automatically generated for the kernel in Figure 8(d).

5. Implementation

We implemented MostoDE in a tool with a graphical user interface to facilitate the automatic generation of SPARQL executable mappings. This tool is based on the Jena framework (Carroll et al., 2004), which allows to work with both RDF and OWL. Furthermore, thanks to the ARQ module, we are able to work with SPARQL queries. Our tool includes two components that allow to deal with the variability of semantic-web technologies regarding query engines and reasoners: the query manager and the reasoner manager. Each of these managers use a RDF store to work with ontologies.

In our implementation, we use two interfaces for the query and reasoner managers to abstract away from the technology. Figure 11 shows the design diagram of these managers, in which we use the Factory and the Adapter design software patterns (Gamma et al., 1995). Furthermore, we have used the following technologies to implement adapters for the query manager: Jena in memory, Oracle, Jena TDB, and Jena SDB. Oracle stores ontologies in an internal representation based on relational tables; Jena TDB supports large scale storage and uses the file system to store ontologies; Jena SDB uses standard SQL databases to store ontologies, such as Microsoft SQL Server, Oracle, IBM DB2, or MySQL. We have also used three technologies to implement adapters for the reasoner manager, namely: Jena, Oracle (OWL Prime), and Pellet. Consequently, this implementation allows to test 4×3 data exchange systems, but our design allows to incorporate new systems easily.



Query manager: Jena In Memory Reasoner manager: Jena

Source Ontology

```
dbp:City [dbp:PopulatedPlace]
dbp:Company [dbp:Organization]
dbp:Organization
    dbp:foundationPlace <dbp:City>
    dbp:foundedBy <dbp:Person>
dbp:Person
    foaf:givenName <xsd:string>
    foaf:surname <xsd:string>
dbp:Place
    dbp:areaTotal <xsd:double>
    geo:lat <xsd:float>
    geo:long <xsd:float>
dbp:PopulatedPlace [dbp:Place]
    dbp:PopulatedPlace/areaTotal <http://dbpedi
```

Correspondences

```
Place ==> location.location (CLASS
areaTotal = dbToFl => location.loca
lat ==> location.geocode.latitude (D
long ==> location.geocode.longitud
areaTotal = sqkmToFl => location.lo
City ==> location.citytown (CLASS)
Organization ==> organization.orga
Company ==> business.employme
foundationPlace ==> organization.c
foundedBy ==> business.employm
Person ==> people.person (CLASS
givenName,surname = concat => ty
```

Target Ontology

```
fb_bus:employment_tenure
    fb_bus:employment_tenure.person <fb_peo:
fb_bus:employment_tenure.company [fb:organiz
    fb_bus:employer.employees <fb_bus:employ
fb_loc:citytown [fb_loc:location]
fb_loc:geocode
    fb_loc:geocode.latitude <xsd:float>
    fb_loc:geocode.longitude <xsd:float>
fb_loc:location
    fb_loc:location.area <xsd:float>
    fb_loc:location.geolocation <fb_loc:geocode>
fb:organization.organization
    fb:organization.organization.place_founded <
fb_peo:person
    fb:type.object.name <xsd:string>
```

Source Restrictions

```
Organization <= dom foundedBy (STRONG_DOMAIN
```

Executable Mappings

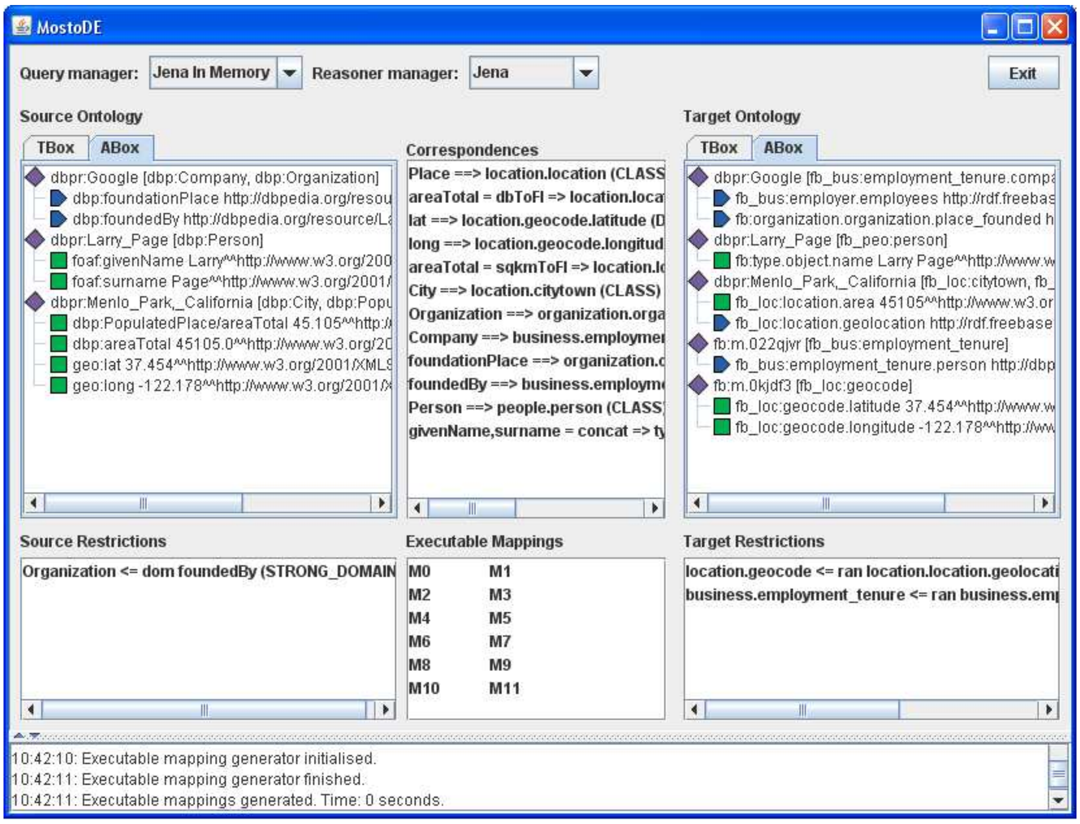
```
M0      M1
M2      M3
M4      M5
M6      M7
M8      M9
M10     M11
```

Target Restrictions

```
location.geocode <= ran location.location.geolocati
business.employment_tenure <= ran business.emp
```

```
10:42:10: Executable mapping generator initialised.
10:42:11: Executable mapping generator finished.
10:42:11: Executable mappings generated. Time: 0 seconds.
```

(a) Sample entities, constraints and correspondences represented in MostoDE.



Query manager: Jena In Memory Reasoner manager: Jena

Source Ontology

```
dbpr:Google [dbp:Company, dbp:Organization]
    dbp:foundationPlace http://dbpedia.org/resou
    dbp:foundedBy http://dbpedia.org/resource/L
dbpr:Larry_Page [dbp:Person]
    foaf:givenName Larry^^http://www.w3.org/200
    foaf:surname Page^^http://www.w3.org/2001/
dbpr:Menlo_Park,_California [dbp:City, dbp:Popu
    dbp:PopulatedPlace/areaTotal 45.105^^http://
    dbp:areaTotal 45105.0^^http://www.w3.org/20
    geo:lat 37.454^^http://www.w3.org/2001/XML
    geo:long -122.178^^http://www.w3.org/2001/
```

Correspondences

```
Place ==> location.location (CLASS
areaTotal = dbToFl => location.loca
lat ==> location.geocode.latitude (D
long ==> location.geocode.longitud
areaTotal = sqkmToFl => location.lo
City ==> location.citytown (CLASS)
Organization ==> organization.orga
Company ==> business.employme
foundationPlace ==> organization.c
foundedBy ==> business.employm
Person ==> people.person (CLASS
givenName,surname = concat => ty
```

Target Ontology

```
dbpr:Google [fb_bus:employment_tenure.compa
    fb_bus:employer.employees http://rdf.freebas
    fb:organization.organization.place_founded h
dbpr:Larry_Page [fb_peo:person]
    fb:type.object.name Larry Page^^http://www.w
dbpr:Menlo_Park,_California [fb_loc:citytown, fb_
    fb_loc:location.area 45105^^http://www.w3.or
    fb_loc:location.geolocation http://rdf.freebase
fb:m.022qjvr [fb_bus:employment_tenure]
    fb_bus:employment_tenure.person http://dbp
fb:m.0kjdf3 [fb_loc:geocode]
    fb_loc:geocode.latitude 37.454^^http://www.w
    fb_loc:geocode.longitude -122.178^^http://ww
```

Source Restrictions

```
Organization <= dom foundedBy (STRONG_DOMAIN
```

Executable Mappings

```
M0      M1
M2      M3
M4      M5
M6      M7
M8      M9
M10     M11
```

Target Restrictions

```
location.geocode <= ran location.location.geolocati
business.employment_tenure <= ran business.emp
```

```
10:42:10: Executable mapping generator initialised.
10:42:11: Executable mapping generator finished.
10:42:11: Executable mappings generated. Time: 0 seconds.
```

(b) Sample source and target data represented in MostoDE.

Figure 12: Two sample screen shots of our tool.

	D2F	O2M	MO	DBP
Classes	13	72	9	12
Data properties	10	41	8	4
Object properties	6	90	8	5
Correspondences	12	11	10	9
Source constraints	20	696	54	12
Target constraints	20	118	58	49
Triples	3,237,758	2,536,567	1,093,928	2,107,451
Executable mappings	12	11	10	9
Generation time (secs)	0.03	0.25	0.08	0.06
Data exchange (secs)	5.46	55.2	20.62	2.95

Table 3: Results of evaluating MostoDE.

Figure 12 presents two screen shots of MostoDE. In Figure 12(a), we show the entities and constraints of the source and target ontologies of our running example. MostoDE segregates the structure of an ontology from its data, see Figure 12(b). Furthermore, the tool allows to define user-defined (source and target) constraints, and correspondences, which are represented in a user-friendly notation. When constraints and correspondences are specified, MostoDE automatically generates SPARQL executable mappings that are also shown to the user.

Below, we describe a number of steps that the user has to perform to exchange data from a source to a target ontology; these steps, which provide an overview on how MostoDE is expected to be used, are the following:

1. The user is responsible for selecting a query and a reasoner manager to store source and target ontologies, perform reasoning, and exchanging data.
2. The user has to choose the source and target ontologies to be integrated, which are automatically parsed and shown to her/him using our tree-like notation.
3. The user is now responsible for adding correspondences amongst source and target entities.
4. In addition to correspondences, it may be necessary to add user-defined constraints because source and target ontologies must be adapted to the requirements of the data exchange problem. The user is responsible for providing them in case that there were necessary; therefore, this step is optional.
5. In this step, the user can instruct MostoDE to automatically generate SPARQL executable mappings using the previously defined constraints and correspondences.
6. Finally, it is also possible to actually exchange data from the source to the target by running the previously generated executable mappings.

6. Evaluation

We have used the previous implementation to evaluate our tool, i.e., to measure the time that it takes to generate executable mappings and perform data exchange. Since timings are imprecise in nature, we repeated each experiment 25 times and averaged the results after discarding roughly 0.01% outliers using the well-known Chevischev's inequality. The experiments were run on a computer that was equipped with a single 2.66 GHz Core 2 Duo CPU and 4GB RAM, Windows XP Professional (SP3), JRE 1.6.0, and Jena 2.6.4. We also measured the time our executable mappings took to exchange data using ARQ 2.8.8 as the query manager and Oracle 11g as the reasoner manager. Although these timings depend largely on the technology being used, we think that presenting them is appealing insofar they prove that the queries we generate can be executed on reasonably-large ontologies in a sensible time.

Table 3 summarises our evaluation results, in which the columns represent the data exchange problems that we have evaluated and the rows a number of measures; the first group of measures provides an overall idea of the size of each data exchange problem, whereas the second group provides information about the number of executable mappings, the time to generate them, and the time they took to execute, i.e., the time of performing the data exchange. Note that the time MostoDE took to generate the executable mappings in the data exchange problems was less than one second in all cases.

The data exchange problems that we have used form part of a benchmark for evaluating data exchange in the context of ontologies (Rivero et al., 2012a). These problems are the following:

- D2F: This problem corresponds to the running example that we present in this article.

- O2M: This problem focuses on publishing semantic web services as Linked Open Data, which is a successful initiative of the Web of Data that consists of a number of principles to publish, connect, and query data in the Web that rely on semantic-web technologies (Heath and Bizer, 2011). OWL-S (Klusch et al., 2009) is one of the main approaches for describing semantic web services that defines an ontology in OWL. MSM (Pedrinaci and Domingue, 2010) is a web service ontology that allows to publish web services as Linked Open Data. In this problem, we exchange data from OWL-S 1.1 to MSM 1.0.

- MO: This problem builds on a fictitious video-on-demand service called Movies Online, which provides information about the movies it broadcasts and reviews of these movies. Movies Online provides an ontology and we wish to exchange data by combining DBpedia 3.2 and Revyu 1.0 (Heath and Motta, 2008), which is a publicly available web site that allows to review and rate web resources, including movies.

- DBP: It focuses on the evolution that an ontology may suffer. DBpedia comprises a number of different versions due to a number of changes in its conceptualisation. When a new version of DBpedia is devised, the new ontology may be populated by performing data exchange from a previous version to the new one. In this problem, we exchange data from a part of DBpedia 3.2 to DBpedia 3.6.

7. Conclusions

In this article, we present MostoDE, a tool that assists software engineers in exchanging data amongst a subset of ontologies that can be represented in quite a complete subset of the OWL 2 Lite profile (see (Rivero et al., 2012b) for further details). It takes a source and a target ontology and a set of correspondences amongst them as input, and it outputs a number of SPARQL executable mappings that are executed by means of a query engine to perform data exchange, and are represented in SPARQL 1.1.

The core of MostoDE comprises an underlying data model that allows to represent constraints, n:1 correspondences, and transformation functions. Additionally, the core comprises a set of algorithms that allows to automatically generate SPARQL executable mappings based on correspondences and constraints amongst source and target ontologies. These algorithms are divided into two groups: a) the generation of kernels, each of which describes the structure of a subset of data in the source ontology that needs to be exchanged as a whole, and the structure of a subset of data in the target ontology that needs to be created as a whole; b) the transformation of kernels into executable mappings in SPARQL.

The key features of our tool are that it allows to automate the generation of executable mappings using correspondences and constraints; it integrates several systems that implement semantic-web technologies to exchange data; and it provides visual aids for helping software engineers to exchange data amongst ontologies. As future work, we are planning to incorporate more complex types of constraints, such as zero-cardinality, general property, or class intersection constraints. Furthermore, to reduce or even avoid the use of reasoners, we are planning on leveraging SPARQL 1.1 entailments once the recommendation is stable.

Acknowledgements

The work that we present in this article was supported by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programmes (grants TIN2007-64119, P07-TIC-2602, P08-TIC-4100, TIN2008-04718-E, TIN2010-21744, TIN2010-09809-E, TIN2010-10811-E, TIN2010-09988-E, and TIN2011-15497-E).

References

Alexe, B., ten Cate, B., Kolaitis, P.G., Tan, W.C., 2011a. Designing and refining schema mappings via data examples, in: SIGMOD Conference, pp. 133–144.

Alexe, B., ten Cate, B., Kolaitis, P.G., Tan, W.C., 2011b. Eirene: Interactive design and refinement of schema mappings via data examples. PVLDB 4, 1414–1417.

Alexe, B., Chiticariu, L., Miller, R.J., Pepper, D., Tan, W.C., 2008a. Muse: A system for understanding and designing mappings, in: SIGMOD, pp. 1281–1284.

Alexe, B., Chiticariu, L., Tan, W.C., 2006. SPIDER: A schema mapping debugger, in: VLDB, pp. 1179–1182.

Alexe, B., Tan, W.C., Velegrakis, Y., 2008b. STBenchmark: Towards a benchmark for mapping systems. PVLDB 1, 230–244.

Altova, 2012. Altova MapForce. `http://www.altova.com/mapforce.html`.

Antoniou, G., van Harmelen, F., 2008. A Semantic Web Primer. The MIT Press.

Bellahsene, Z., Bonifati, A., Rahm, E. (Eds.), 2011. Schema Matching and Mapping. Springer.

Bernstein, P.A., Melnik, S., 2007. Model management 2.0: Manipulating richer mappings, in: SIGMOD, pp. 1–12.

Bikakis, N., Gioldasis, N., Tsinaraki, C., Christodoulakis, S., 2009. Querying XML data with SPARQL, in: DEXA, pp. 372–381.

Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., Hellmann, S., 2009. DBpedia: A crystallization point for the Web of Data. J. Web Sem. 7, 154–165.

Bizer, C., Schultz, A., 2010. The R2R framework: Publishing and discovering mappings on the web, in: COLD.

Bollacker, K.D., Evans, C., Paritosh, P., Sturge, T., Taylor, J., 2008. Freebase: a collaboratively created graph database for structuring human knowledge, in: SIGMOD Conference, pp. 1247–1250.

Bonifati, A., Chang, E.Q., Ho, T., Lakshmanan, L.V.S., Pottinger, R., 2005. HePToX: Marrying XML and heterogeneity in your P2P databases, in: VLDB, pp. 1267–1270.

Bouquet, P., Giunchiglia, F., van Harmelen, F., Serafini, L., Stuckenschmidt, H., 2004. Contextualizing ontologies. J. Web Sem. 1, 325–343.

Carey, M.J., 2006. Data delivery in a service-oriented world: The BEA AquaLogic data services platform, in: SIGMOD Conference, pp. 695–705.

Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K., 2004. Jena: Implementing the semantic web recommendations, in: WWW, pp. 74–83.

Dou, D., McDermott, D.V., Qi, P., 2005. Ontology translation on the Semantic Web. J. Data Semantics 2, 35–57.

Euzenat, J., Polleres, A., Scharffe, F., 2008. Processing ontology alignments with SPARQL, in: CISIS, pp. 913–917.

Euzenat, J., Shvaiko, P., 2007. Ontology matching. Springer-Verlag.

Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L., 2005. Data exchange: Semantics and query answering. Theor. Comput. Sci. 336, 89–124.

Fensel, D., Lausen, H., Polleres, A., de Bruijn, J., Stollberg, M., Roman, D., Domingue, J., 2007. Enabling Semantic Web Services: The Web Service Modeling Ontology. Springer.

Forte, M., de Souza, W.L., do Prado, A.F., 2008. Using ontologies and web services for content adaptation in ubiquitous computing. Journal of Systems and Software 81, 368–381.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Gennari, J.H., Musen, M.A., Fergerson, R.W., Grosso, W.E., Crubézy, M., Eriksson, H., Noy, N.F., Tu, S.W., 2003. The evolution of Protégé: An environment for knowledge-based systems development. Int. J. Hum.-Comput. Stud. 58, 89–123.

Glimm, B., Hogan, A., Krötzsch, M., Polleres, A., 2012. OWL: Yet to arrive on the Web of Data?, in: LDOW.

Haarslev, V., Möller, R., 2008. On the scalability of description logic instance retrieval. J. Autom. Reasoning 41, 99–142.

Haas, L.M., Hernández, M.A., Ho, H., Popa, L., Roth, M., 2005. Clio grows up: From research prototype to industrial tool, in: SIGMOD, pp. 805–810.

Haase, P., Lewen, H., Studer, R., Tran, D.T., Erdmann, M., d'Aquin, M., Motta, E., 2008. The NeOn ontology engineering toolkit, in: WWW.

Heath, T., Bizer, C., 2011. Linked Data: Evolving the Web into a Global Data Space. Morgan & Claypool.

Heath, T., Motta, E., 2008. Revyu: Linking reviews and ratings into the Web of Data. J. Web Sem. 6, 266–273.

IBM, 2012. IBM's InfoSphere Data Architect. `http://www-01.ibm.com/software/data/optim/data-architect/`.

Klusch, M., Fries, B., Sycara, K.P., 2009. OWLS-MX: A hybrid semantic web service matchmaker for OWL-S services. J. Web Sem. 7, 121–133.

Lee, J., Park, J.H., Park, M.J., Chung, C.W., Min, J.K., 2010. An intelligent query processing for distributed ontologies. Journal of Systems and Software 83, 85–95.

Lenzerini, M., 2002. Data integration: A theoretical perspective, in: PODS, pp. 233–246.

Maedche, A., Motik, B., Silva, N., Volz, R., 2002. MAFRA: A mapping framework for distributed ontologies, in: EKAW, pp. 235–250.

Makris, K., Bikakis, N., Gioldasis, N., Christodoulakis, S., 2012. SPARQL-

RW: Transparent query access over mapped RDF data sources, in: EDBT, pp. 610–613.

Makris, K., Gioldasis, N., Bikakis, N., Christodoulakis, S., 2010. Ontology mapping and SPARQL rewriting for querying federated RDF data sources, in: ODBASE, pp. 1108–1117.

Marnette, B., Mecca, G., Papotti, P., Raunich, S., Santoro, D., 2011. ++Spicy: An open-source tool for second-generation schema mapping and data exchange. PVLDB 4, 1438–1441.

Mecca, G., Papotti, P., Raunich, S., 2009a. Core schema mappings, in: SIGMOD Conference, pp. 655–668.

Mecca, G., Papotti, P., Raunich, S., Buoncristiano, M., 2009b. Concise and expressive mappings with +Spicy. PVLDB 2, 1582–1585.

Microsoft, 2012. Microsoft BizTalk Mapper. `http://msdn.microsoft.com/en-us/library/aa547076.aspx`.

Mocan, A., Cimpian, E., 2007. An ontology-based data mediation framework for semantic environments. Int. J. Semantic Web Inf. Syst. 3, 69–98.

Mokhtar, S.B., Preuveneers, D., Georgantas, N., Issarny, V., Berbers, Y., 2008. EASY: Efficient semantic service discovery in pervasive computing environments with QoS and context support. Journal of Systems and Software 81, 785–808.

Motik, B., Horrocks, I., Sattler, U., 2009. Bridging the gap between OWL and relational databases. J. Web Sem. 7, 74–89.

Noy, N.F., 2004. Semantic integration: A survey of ontology-based approaches. SIGMOD Record 33, 65–70.

Noy, N.F., Klein, M.C.A., 2004. Ontology evolution: Not the same as schema evolution. Knowl. Inf. Syst. 6, 428–440.

Noy, N.F., Musen, M.A., 2003. The PROMPT suite: interactive tools for ontology merging and mapping. Int. J. Hum.-Comput. Stud. 59, 983–1024.

Omelayenko, B., 2002. Integrating vocabularies: Discovering and representing vocabulary maps, in: ISWC, pp. 206–220.

Parreiras, F.S., Staab, S., Schenk, S., Winter, A., 2008. Model driven specification of ontology translations, in: ER, pp. 484–497.

Pedrinaci, C., Domingue, J., 2010. Toward the next wave of services: Linked services for the Web of Data. J. UCS 16, 1694–1719.

Petropoulos, M., Deutsch, A., Papakonstantinou, Y., Katsis, Y., 2007. Exporting and interactively querying web service-accessed sources: The CLIDE system. ACM Trans. Database Syst. 32.

Pichler, R., Savenkov, V., 2009. DEMo: Data exchange modeling tool. PVLDB 2, 1606–1609.

Polleres, A., Huynh, D., 2009. Special issue: The Web of Data. J. Web Sem. 7, 135.

Polleres, A., Scharffe, F., Schindlauer, R., 2007. SPARQL++ for mapping between RDF vocabularies, in: ODBASE, pp. 878–896.

Popa, L., Velegrakis, Y., Miller, R.J., Hernández, M.A., Fagin, R., 2002. Translating web data, in: VLDB, pp. 598–609.

Raffio, A., Braga, D., Ceri, S., Papotti, P., Hernández, M.A., 2008a. Clip: A tool for mapping hierarchical schemas, in: SIGMOD Conference, pp. 1271–1274.

Raffio, A., Braga, D., Ceri, S., Papotti, P., Hernández, M.A., 2008b. Clip: A visual language for explicit schema mappings, in: ICDE, pp. 30–39.

Ressler, J., Dean, M., Benson, E., Dorner, E., Morris, C., 2007. Application of ontology translation, in: ISWC, pp. 830–842.

Rivero, C.R., Hernández, I., Ruiz, D., Corchuelo, R., 2011a. Generating SPARQL executable mappings to integrate ontologies, in: ER, pp. 118–131.

Rivero, C.R., Hernández, I., Ruiz, D., Corchuelo, R., 2011b. Mosto: Generating SPARQL executable mappings between ontologies, in: ER Workshops, pp. 345–348.

Rivero, C.R., Hernández, I., Ruiz, D., Corchuelo, R., 2012a. Benchmarking data exchange amongst semantic-web ontologies. IEEE Trans. Knowl. Data Eng. Preprint.

Rivero, C.R., Hernández, I., Ruiz, D., Corchuelo, R., 2012b. Exchanging data amongst Linked Data applications. Knowl. Inf. Sys. Preprint.

Serafini, L., Tamilin, A., 2007. Instance migration in heterogeneous ontology environments, in: ISWC, pp. 452–465.

SPARQL, 2012. SPARQL 1.1 test results. `http://www.w3.org/2009/sparql/implementations/`.

Stylus, 2012. Stylus Studio. `http://www.stylusstudio.com/`.

W3C, 2012. W3C tools. `http://www.w3.org/2001/sw/wiki/Tools`.

Zhang, H., Li, Y.F., Tan, H.B.K., 2010. Measuring design complexity of semantic web ontologies. Journal of Systems and Software 83, 803–814.